PROJETO FINAL

ALUNO(A):_____NOTA:_____

ATENÇÃO: Descrever as soluções com o máximo de detalhes possível, no caso de programas, inclusive a forma como os testes foram feitos. Todos os artefatos (relatório, código fonte de programas, e outros) gerados para este trabalho devem ser adicionados em um repositório no site `github.com`, com o seguinte formato:

`FinalProject_OS_UFRR_Desc_X_2025`.

[DESCRIÇÃO - 1]
Cliente/Servidor Seguro com Compactação e Criptografia TLS

Neste projeto, você irá desenvolver uma aplicação cliente-servidor baseada em terminal remoto (semelhante ao Telnet), com suporte a comunicação segura (via TLS usando a biblioteca OpenSSL) e comunicação eficiente (via compactação de dados com zlib). A proposta é permitir o uso remoto de um shell, tratando os desafios da comunicação em rede sem modificar a lógica do shell original, por meio da construção de agentes intermediários (cliente e servidor) que encapsulam as complexidades de segurança e desempenho.

Quando uma aplicação como um shell é acessada remotamente, simplesmente redirecionar a entrada e saída padrão para um socket TCP não é suficiente. Sessões remotas exigem tratamento especial, como criptografia, controle de sinais (como `^C` e `^D`), e otimização do tráfego. Para isso, aplicações modernas utilizam proxies cliente/servidor que intermediam a comunicação entre o shell e o terminal do usuário, adicionando segurança (TLS), compressão de dados (zlib) e registro de atividade (logs), de forma transparente para o usuário final.

Você irá construir dois programas principais:

- **Cliente**: conecta-se ao servidor via socket TCP/TLS, captura a entrada do teclado e envia para o servidor. Exibe na tela a resposta recebida.
- **Servidor**: aceita conexões do cliente, inicia um shell (bash ou sh) como subprocesso e atua como intermediador entre o cliente e o shell.

A comunicação entre cliente e servidor pode ser:

- **Simples** (via TCP),
- **Compactada** (com zlib), e/ou
- **Segura** (com TLS via OpenSSL).

Funcionalidades do Cliente

- **Conexão TCP**: O cliente deve se conectar ao servidor especificando a porta com `--port=NUMERO` e opcionalmente o host com `--host=ENDERECO`.
- **Transmissão de Entrada/Saída**: Entrada do teclado vai para o socket; resposta recebida é impressa no terminal.
- **Log**: Com a opção `--log=arquivo.log`, o cliente registra todos os dados enviados e

recebidos no formato:

```
SENT 35 bytes: <dados enviados>
RECEIVED 18 bytes: <dados recebidos>
```

- **Compactação**: Ao usar `--compress`, todos os dados trocados devem ser compactados antes do envio e descompactados após o recebimento.
- **Criptografia TLS (OpenSSL)**: Com a opção `--encrypt`, a conexão deve ser segura via TLS. Utilize a biblioteca OpenSSL para estabelecer o canal criptografado.

Funcionalidades do Servidor

- **Escuta em Porta TCP**: Deve aceitar conexões usando `--port=NUMERO`.
- **Criação do Shell**: Ao aceitar uma conexão, crie um processo filho (`fork`) que inicia um shell. Utilize `pipe()` para se comunicar com ele.
- **Encaminhamento de Dados**:
  - Entrada do cliente → shell (via pipe)
  - Saída do shell → cliente (via socket)
- **Tratamento de Sinais**:
  - Se o cliente enviar `^C`, converta em `SIGINT` para o shell.
  - Ao detectar `^D`, feche a escrita no pipe (EOF).
- **Compactação e TLS**: As opções `--compress` e `--encrypt` devem funcionar conforme descritas para o cliente.

Componentes Técnicos

- **Sockets TCP**: Comunicação entre cliente e servidor.
- **OpenSSL**: Para implementar o canal criptografado com TLS. Requer uso de funções como `SSL_new`, `SSL_connect`, `SSL_read`, `SSL_write`, etc.
- **zlib**: Para compressão e descompressão de dados.
- **Pipes** e **fork/exec**: Para iniciar e controlar o shell no servidor.

Exemplo de Execução

Servidor:

```
./servidor --port=12345 --compress --encrypt
```

Cliente:

```
./cliente --port=12345 --host=localhost --log=transmissao.log --compress --encrypt
```

Requisitos Adicionais

- A saída do cliente deve continuar funcionando mesmo se a conexão for reiniciada (tratar exceções).
- Os logs devem registrar os dados exatamente **após compactação** e **antes de descompactação**.
- Os arquivos de log devem ser incluídos na entrega final do projeto.
- Certifique-se de lidar corretamente com buffers parciais, fim de fluxo, e erros.

Referências Úteis

- Sockets TCP: http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
- Compressão com zlib: https://www.zlib.net/zlib_how.html
- Guia OpenSSL: https://wiki.openssl.org/index.php/SSL/TLS_Client

[DESCRIÇÃO - 2]

Desenvolvimento de um mini terminal Linux, ou seja, um interpretador de comandos, o qual deverá aceitar a maioria dos comandos convencionais do terminal Linux, além da criação de processos e tratamento dos mesmos.

Neste trabalho você deve implementar um interpretador de comandos (chamado no Unix/Linux de *shell*). Para isso, você deverá aprender como disparar processos e como encadear sua comunicação usando *pipes*. Além disso, você deverá utilizar pipes e manipulação de processos para criar um par produtor/consumidor por envio de mensagens que trabalhará dentro da *shell* para ``alimentar'' programas que você disparar com dados, ou para receber os dados de um programa.

Seu programa deverá ser implementado em C (não serão aceitos recursos de C++, como classes da biblioteca padrão ou de outras fontes). O programa, que deverá se chamar `shellso` deverá ser iniciado sem argumentos de linha de comando, ou com apenas um. Caso seja disparado sem argumentos, ele deverá escrever um *prompt* no início de cada linha na tela (um símbolo como “>>PROMPT ” ou uma mensagem como “Sim, mestre?”) e depois ler comandos da sua entrada padrão (normalmente, o teclado). Mensagens de erro e o resultado dos programas, salvo quando redirecionados pelos comandos fornecidos, devem ser exibidos na saída padrão (usualmente, a janela do terminal). Essa forma de operação é denominada interativa. Já se um parâmetro for fornecido, ele deve ser interpretado como o nome de um arquivo de onde comandos serão lidos. Nesse caso, apenas o resultado dos comandos deve ser exibido na saída padrão, sem a exibição de *prompts* nem o nome dos comandos executados.

Em ambos os modos de operação, sua *shell* termina ao encontrar o comando “`fim`” no início de uma linha ou ao encontrar o final do fluxo de bytes de entrada (ao fim do arquivo de entrada, ou se o usuário digita `Ctrl-D` no teclado). Cada linha deve conter um comando ou mais comandos para ser(em) executado(s). No caso de haver mais de um programa a ser executado na linha, eles devem obrigatoriamente ser encadeados por *pipes* (“|”), indicando que a saída do programa à esquerda deve ser associada à entrada do programa à direita.

Um novo *prompt* só deve ser exibido (se necessário) e um novo comando só deve ser lido quando o comando da linha anterior terminar sua execução, exceto caso a linha de comando termine com um "&", quando o programa deve ser executado em *background*. Em qualquer caso, o interpretador deve sempre receber o valor de saída dos programas executados -- isto é, ele não deve deixar zumbis no sistema (confira as chamadas de sistema `wait()` e `waitpid()` para esse fim). Para o caso dos programas executados em *background*, você deve se informar sobre o tratamento de sinais, em particular o sinal SIGCHLD, para tratar o término daqueles programas.

O seu interpretador não aceitará os comandos de redirecionamento de entrada e saída normalmente utilizados, "<" e ">". Ao invés disso, ele aceitará os símbolos "=>" e "<=", que indicarão entrada e saída por *pipes* para processos produtores/consumidores.

Com base nessa descrição, são comandos válidos (supondo que o *prompt* seja "Qual o seu comando?":

```
Qual o seu comando? ls -l
Qual o seu comando? ls -laR => arquivo
Qual o seu comando?
Qual o seu comando? wc -l <= arquivo &
Qual o seu comando? cat -n <= arquivo => arquivonumerado
Qual o seu comando? cat -n <= arquivo | wc -l => numerodelinhas
Qual o seu comando? cat -n <= arquivo | wc -l => numerodelinhas &
Qual o seu comando? fim
```

Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

1. Descreva as principais funções do terminal;
2. Descrever as funções dos seguintes componentes do terminal: parser; executor; subsistemas do terminal;
3. Descrever a implementação para a criação de processos;
   - No seu interpretador crie pelo menos um novo processo para cada novo comando;
   - Para ler linhas da entrada, você pode querer olhar a função `fgets()`;
4. Descrever a implementação de pipe e redirecionamento no terminal;
5. Descrever a implementação de wildcards no terminal;
6. Codificação de caracteres no terminal; e
7. Lista de comandos disponível;

[DESCRIÇÃO – 3]
Criação de um Sistema de Arquivos Virtual de 8 bits (Toy File System)

Você será responsável por projetar e implementar seu próprio sistema de arquivos (toy file system), que deverá conter funcionalidades básicas como leitura, escrita e estruturação hierárquica (árvore de diretórios). O projeto utilizará o conceito de sistema de arquivos virtual (VFS – *Virtual File System*), possibilitando testes e simulações no Linux. A proposta inclui uma etapa de estudo, experimentação com exemplos existentes e desenvolvimento de uma solução própria. Etapas de construção:

1. Estudo Inicial:

- **Leitura Obrigatória:**
  Estude o artigo "Creating Linux virtual filesystems", disponível no portal LWN.net.

2. Análise de Sistemas de Arquivos Existentes:

- **Explore e teste os seguintes sistemas de arquivos de exemplo:**
  - Gogislenefs
  - Hellofs

  ➤ Descreva o funcionamento básico de cada exemplo, principais estruturas utilizadas e funcionalidades implementadas.

3. Desenvolvimento do seu Toy File System:

- **Requisitos mínimos:**
  - Implementar operações básicas de **leitura** e **escrita** de arquivos;
  - Os arquivos podem ser mantidos em memória (não é necessário persistência em disco);
  - Implementar uma **estrutura hierárquica de diretórios**, com suporte à criação, navegação e remoção;
  - Utilizar bibliotecas conhecidas como **FUSE (Filesystem in Userspace)**, e linguagens como C ou C++;
  - Simular o sistema de arquivos usando uma imagem de disco ou pendrive virtualizado.
- **Diferenciais recomendados (opcional):**
  - Suporte a permissões de acesso;
  - Implementação de blocos e inodes simplificados;
  - Ferramenta de montagem e desmontagem via terminal.

4. Execução e Testes Utilizando um Pendrive

Para testar o seu sistema de arquivos em um ambiente realista, utilize um **pendrive formatado com um sistema de arquivos padrão** (por exemplo, FAT32 ou ext4) apenas como meio de armazenamento temporário. A montagem e execução do seu File System ocorrerá sobre um arquivo ou partição neste pendrive, sem necessidade de sobrescrever seu sistema nativo.

Procedimentos recomendados:

1. **Montagem do pendrive** em `/mnt/usb`:
   ```
   sudo mount /dev/sdX1 /mnt/usb
   ```
2. **Criação de um arquivo de imagem dentro do pendrive** (simulando um disco virtual):
   ```
   dd if=/dev/zero of=/mnt/usb/toyfs.img bs=1M count=10
   ```
3. **Formatação da imagem com seu sistema de arquivos personalizado (via script)**.
4. **Montagem via FUSE:**
   ```
   mkdir /mnt/toyfs
   ./toyfs /mnt/toyfs /mnt/usb/toyfs.img
   ```
5. **Testes de leitura, escrita e criação de diretórios**:
   ```
   echo "teste" > /mnt/toyfs/arquivo.txt
   ```

```
cat /mnt/toyfs/arquivo.txt
mkdir /mnt/toyfs/teste_dir
```

6. **Desmontagem após os testes:**

```
fusermount -u /mnt/toyfs
```

⚠️ *Certifique-se de não sobrescrever dados importantes do pendrive. Use apenas arquivos dentro do pendrive como armazenamento virtual.*

[DESCRIÇÃO – 4]
Simulador de memória virtual e física

Será implementado um simulador de memória virtual e física. Seu programa deverá ser implementado em C (não serão aceitos recursos de C++, como classes da biblioteca padrão ou de outras fontes).

O simulador receberá **como entrada um arquivo** que conterá a sequência de endereços de memória acessados por um programa real (na verdade, apenas uma parte da sequência total de acessos de um programa). Esses endereços estarão escritos como **números hexadecimais**, seguidos por uma letra R ou W, para indicar se o acesso foi de leitura ou escrita. Ao iniciar o programa, será definido o tamanho da memória (em quadros) para aquele programa e qual o algoritmo de substituição de páginas a ser utilizado. O programa deve, então, processar cada acesso à memória para atualizar os bits de controle de cada quadro, detectar falhas de páginas (*page faults*) e simular o processo de carga e substituição de páginas. Durante todo esse processo, estatísticas devem ser coletadas, para gerar um relatório curto ao final da execução.

O programa, que deverá ser iniciado com quatro argumentos, exemplo, `progvirtual <polisub> <arquivo.log> <sizePg> <sizeFis>.` Esses argumentos representam, pela ordem:

1. O `<polisub>` algoritmo de substituição a ser usado (`lru`, `fifo` ou `random`);
2. O `<arquivo.log>` arquivo contendo a sequência de endereços de memória acessados;
3. O `<sizePg>` é o tamanho de cada página/quadro de memória, em kilobytes -- faixa de valores razoáveis: de 2 a 64;
4. O `<sizeFis>` é o tamanho total da memória física disponível para o processo, também em kilobytes - faixa de valores razoáveis: de 128 a 16384 (16 MB).

Formato da saída do simulador. Ao final da simulação, quando a sequência de acessos à memória terminar, o programa deve gerar um pequeno relatório, contendo:

- a configuração utilizada (definida pelos quatro parâmetros);
- o número total de acessos à memória contidos no arquivo;
- o número de page faults (páginas lidas); e
- o número de páginas “sujas” que tiveram que ser escritas de volta no disco (lembrando-se que páginas sujas que existam no final da execução não precisam ser escritas).

Um exemplo de saída poderia ser da forma (valores completamente fictícios):

```
prompt> simvirtual lru arquivo.log 4 128
Executando o simulador...
Arquivo de entrada: arquivo.log
```

```
Tamanho da memoria: 128 KB
Tamanho das páginas: 4 KB
Tecnica de reposicao: lru
Paginas lidas: 520
Paginas escritas: 352
```

Mais detalhes podem ser consultados em:
https://homepages.dcc.ufmg.br/~dorgival/cursos/so/tp3.html

[DESCRIÇÃO – 5]
Virtualização de Sistemas Operacionais – VB e Containers

Neste projeto o seu objetivo é criar uma rede de computadores virtuais que se comuniquem usando Máquinas Virtuais (exemplo, Virtual Box) e com Containers (exemplo, Docker). Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

- Processo para instalação de um software para a criação de máquinas virtuais e containers;
- Criar um cluster com máquinas virtuais;
- Criar um cluster de containers;
- Apresentar uma avaliação de vantagens e desvantagens entre máquinas virtuais e containers;
- Desenvolva um sistema web escalável usando containers, sendo que cada aspecto do sistema deve conter containers para objetivos específicos, exemplo, um container para o banco de dados, servidor web e outros que se façam necessário. Adicionalmente, aplique um software de sua escolha para a gerência dos containers; e
- Apresente 3 sistemas operacionais com o foco em virtualização de sistemas operacionais.

[DESCRIÇÃO – 6]
Scheduling Policy to the Linux Kernel

O escalonador de processos é o núcleo do sistema operacional. É responsável por escolher um novo processo para ser executado sempre que a CPU estiver disponível. Você estudará partes do escalonador do Linux em profundidade e entenderá como ele decide qual processo executar. Você modificará o escalonador do Linux para implementar uma nova classe de escalonador, chamada de escalonador em background. Você avaliará o desempenho de sua nova classe de escalonador.

A nova política de agendamento chamada SCHED_BACKGROUND, projetada para suportar processos que só precisam ser executados quando o sistema não tem mais nada a fazer. Essa política de agendamento "em segundo plano" só executa processos quando não há processos nas classes SCHED_OTHER, SCHED_RR ou SCHED_FIFO a serem executados. Quando houver mais de um processo SCHED_BACKGROUND pronto para ser executado, eles devem competir pela CPU, assim como os processos SCHED_OTHER.

Neste projeto você irá testar políticas de escalonamento e implementar uma nova no kernel do linux. Com base na descrição apresentada, para o referido projeto deve ser apresentado os seguintes itens:

- Descrever o funcionamento de um escalonamento;
- Criar um tutorial com exemplos de códigos para a criação de uma política de escalonamento no linux;
- Apresente 4 políticas de escalonamento suportadas pelo kernel do linux; e
- Depois de implementar sua política de agendamento e depurá-la com cuidado, você avaliará seu desempenho executando combinações de processos intensivos de CPU em diferentes políticas de agendamento e medindo o tempo necessário;
  - Você pode usar o `gettimeofday()` para medir o tempo "wallclock" e `getrusage()` para registrar o tempo do usuário e do sistema. Sua análise deve incluir wallclock, sistema e tempo do usuário em milissegundos.

Segue abaixo alguns links:
https://helix979.github.io/jkoo/post/os-scheduler/

[DESCRIÇÃO – 7]
Simulação e Teste de Driver de Caractere (LKM) para Linux com Uso de Pendrive Bootável e Emuladores

Imagine um cenário onde uma equipe de técnicos precisa coletar informações personalizadas de sensores conectados a um equipamento industrial via interface serial. Para essa tarefa, foi desenvolvido um driver de caractere personalizado que coleta e interpreta os dados serializados recebidos por um dispositivo específico (simulado no projeto). Este driver precisa ser carregado de forma dinâmica em diversos ambientes Linux portáteis, diretamente de um **pendrive bootável**.

Esse projeto simula exatamente esse tipo de situação, fornecendo ao aluno a oportunidade de desenvolver, carregar e testar o driver em um ambiente Linux controlado, como:

- Uma **distribuição Linux leve instalada em um pendrive bootável** (ex: Debian Live, Ubuntu Server, Alpine);
- Um sistema Linux **simulado com QEMU** carregado a partir do pendrive;
- Possibilidade de uso de ferramentas como **Ventoy** para múltiplas ISOs Linux no mesmo pendrive.

Descrição das Etapas do Projeto:

1. Preparação do Ambiente no Pendrive:

- Criar um pendrive bootável com uma distribuição Linux leve com suporte ao build de módulos do kernel;
- Instalar (ou embutir via chroot) as ferramentas necessárias: `gcc`, `make`, `modprobe`, `insmod`, `dmesg`, `lsmod`, `rmmod`, e headers do kernel;
- (Opcional) Incluir QEMU para simulação adicional ou executar diretamente no hardware a partir do pendrive.

2. Desenvolvimento do Driver:

- Criar um módulo LKM do tipo *character device* que simule, por exemplo, um "sensor virtual" que gera dados aleatórios quando lido via `/dev/sensor0`;
- Implementar funções de leitura (`read`) e escrita (`write`) simples, com logs em `dmesg` para facilitar os testes.

3. Compilação e Carregamento Dinâmico:

- Gerar o `.ko` do módulo no próprio pendrive ou compilá-lo previamente e copiar para o sistema;
- Carregar o driver utilizando o comando:

```
sudo modprobe sensor_driver   # Se houver suporte a aliases/modprobe.d
```

ou

```
sudo insmod sensor_driver.ko
```

- Criar o nó de dispositivo com:

```
sudo mknod /dev/sensor0 c <major> 0
sudo chmod 666 /dev/sensor0
```

- Verificar com `lsmod`, `dmesg` e testar:

```
echo "123" > /dev/sensor0
cat /dev/sensor0
```

4. Execução em Ambiente Emulado (opcional):

- Usar QEMU para simular uma máquina Linux no próprio pendrive:

```
qemu-system-x86_64 -m 512 -kernel bzImage -initrd initrd.img -append
"root=/dev/sda1"
```

5. Remoção e Teste de Persistência:

- Usar `rmmod` ou `modprobe -r` para descarregar o módulo;
- Testar persistência de logs e comportamento após reinicialização.

Resultados Esperados:

- Um driver funcional capaz de simular um dispositivo virtual em `/dev/sensor0`;
- Testes demonstrando leitura/escrita no driver com ferramentas básicas;
- Scripts de automação no pendrive para carregar e testar o driver com `modprobe`;
- Documentação com prints de `dmesg`, `lsmod`, `cat /dev/sensor0`, etc.

[DESCRIÇÃO – 8]
Desenvolvimento e Simulação de um Gerenciador de Boot Inspirado no GRUB com QEMU

O gerenciador de boot é uma peça essencial no processo de inicialização de qualquer sistema operacional. O GRUB é amplamente utilizado em sistemas Linux e seu funcionamento é um excelente ponto de partida para entender a transição entre o firmware (BIOS/UEFI) e o sistema operacional. No entanto, a complexidade do GRUB real pode ser um obstáculo ao aprendizado. Por isso, o projeto propõe que os alunos construam um gerenciador de boot simplificado, que atenda funcionalidades básicas e obrigatórias, utilizando ambientes controlados e simulados por QEMU.

Objetivos do Projeto:

1. **Compreender o funcionamento básico do processo de boot.**
2. **Utilizar o QEMU para simular arquiteturas x86 e testar imagens de disco.**
3. **Projetar e implementar um gerenciador de boot em Assembly e/ou C.**
4. **Integrar e testar o gerenciador com um kernel simples em um sistema de arquivos virtual.**
5. **Implementar obrigatoriamente ao menos um dos seguintes requisitos avançados:**
    - Um **menu de boot** com seleção interativa;
    - **Leitura de arquivos FAT12 ou FAT16** para carregar o kernel;
    - **Suporte a múltiplos kernels**, com base na escolha do usuário no menu.

Requisitos Técnicos:

- Ambiente Linux (ou WSL para Windows).
- Ferramentas: `QEMU`, `NASM`, `GCC`, `dd`, `mkfs.fat`, `mount`, `losetup`, `objcopy`.
- Linguagens: Assembly (NASM) e/ou C.
- Familiaridade com conceitos como endereçamento de memória, MBR, interrupções e chamadas de BIOS.

Etapas Sugeridas do Projeto:

1. **Estudo Teórico:** Revisão do processo de boot e funcionamento do GRUB.
2. **Configuração do Ambiente:** Instalação e configuração do QEMU e ferramentas auxiliares.
3. **Criação de Imagem de Disco Virtual:** Com MBR e sistema de arquivos FAT12/FAT16.
4. **Bootloader (1º Estágio):** Código simples em Assembly que carregue o 2º estágio.
5. **Gerenciador de Boot (2º Estágio):**
    - Exibir menu de boot (se implementado);
    - Acessar arquivos do disco (se FAT12/16 implementado);
    - Carregar um ou mais kernels conforme opção do usuário.
6. **Desenvolvimento do Kernel:** Criar kernel(s) simples em C com funcionalidades mínimas

(ex.: printar texto, rodar loop).

7. **Execução e Testes:** Com `qemu-system-x86_64`, simulando o processo de boot e escolha de kernel.
8. **Documentação Técnica:** Explicar decisões de projeto, funcionamento e limitações.

[DESCRIÇÃO – 9]
Implementação de um "Mini-Sudo" em Linux com Rust e QEMU

Desenvolver uma versão simplificada do comando `sudo`, utilizando a linguagem de programação **Rust**, e testá-la em um ambiente isolado baseado no emulador **QEMU**. O projeto visa familiarizar os alunos com conceitos de segurança, controle de permissões no Linux e desenvolvimento de software seguro em Rust.

Descrição do Projeto

O utilitário `sudo` permite que usuários executem comandos com privilégios de superusuário. Neste projeto, os alunos irão implementar uma versão básica deste utilitário, chamada `minisudo`, que:

- Valida a identidade do usuário;
- Solicita a senha;
- Executa comandos com privilégios elevados (simulados);
- Registra logs de uso;
- Impede escalonamento de privilégios indevido.

O programa será desenvolvido inteiramente em **Rust**, utilizando as boas práticas de segurança da linguagem (ex. borrowing, ownership, match, pattern safety). O sistema será testado dentro de uma máquina virtual Linux criada com **QEMU**, utilizando uma imagem leve (por exemplo, Alpine ou Debian minimal).

Funcionalidades Mínimas Esperadas

1. **Autenticação simples**:
   - Verificação do nome do usuário e senha via leitura de um arquivo seguro (`/etc/minisudoers` simulado);
   - Hashing e verificação de senha usando a biblioteca `argon2` ou `bcrypt` de Rust.
2. **Execução de comandos simulados**:
   - A execução real com `root` poderá ser simulada com mensagens de log para evitar riscos;
   - Caso seja possível, implementar com uso de `setuid` ou chamada ao `pkexec`.
3. **Registro de uso**:
   - Criar um log de uso (`/var/log/minisudo.log`) com horário, comando e usuário.
4. **Testes automatizados no ambiente QEMU**:

- Scripts de provisionamento e testes que simulam o uso do `minisudo`;
- Utilização de `cargo test` para validar funcionalidades básicas.

Requisitos Técnicos

- Linguagem: Rust (com uso de crates como `clap`, `rpassword`, `bcrypt` ou `argon2`)
- Ambiente de Teste: QEMU com imagem Linux mínima (Alpine, Debian, Arch ou customizada)
- Ferramentas: `cargo`, `make`, `qemu-system-x86_64`, `expect` (para automação de testes)

[DESCRIÇÃO – 10]
Análise de Estratégias de Escalonamento em Sistemas de Tempo Real com Zephyr RTOS

Explorar, implementar e analisar diferentes estratégias de escalonamento de tarefas em sistemas de tempo real utilizando exclusivamente o **Zephyr RTOS**. A atividade busca desenvolver competências práticas no uso de um RTOS moderno e modular, destacando o comportamento do *kernel* sob políticas como **preemptiva**, **cooperativa**, **prioridade fixa** e **Round Robin**.

Os alunos deverão desenvolver um sistema embarcado, em hardware real (como **ESP32**, **STM32**) ou emulador (**QEMU**), com **três tarefas concorrentes** representando funções típicas de um sistema embarcado. As tarefas possuem diferentes prioridades e frequências de execução, sendo:

- **Tarefa 1 – Leitura de Sensor (Alta prioridade)**
  Executa a cada **500 ms**, simulando a coleta de dados de um sensor crítico.
- **Tarefa 2 – Comunicação com Teclado (Média prioridade)**
  Executa a cada **1000 ms**, simulando a leitura de entradas do usuário via teclado ou botão.
- **Tarefa 3 – Atualização de Display (Baixa prioridade)**
  Executa a cada **2000 ms**, simulando a renderização ou atualização visual de uma interface.

O objetivo é observar o comportamento do sistema sob diferentes estratégias de escalonamento oferecidas pelo Zephyr.

Etapas da Atividade

1. **Leitura Preparatória**
   - Leitura da documentação oficial do Zephyr sobre gerenciamento de *threads* e *scheduling*.
2. **Configuração do Ambiente**
   - Instalação do Zephyr RTOS, SDK, `west`, toolchain e dependências.
   - Configuração de ambiente de desenvolvimento (VSCode, CLI, etc).
   - Escolha de hardware ou simulador (ESP32, STM32, QEMU...).

3. **Desenvolvimento do Sistema**
    - Criar as **três tarefas (threads)** com prioridades e períodos distintos:
        - Tarefa 1: prioridade **alta** (`K_PRIO_PREEMPT(0)`)
        - Tarefa 2: prioridade **média** (`K_PRIO_PREEMPT(1)`)
        - Tarefa 3: prioridade **baixa** (`K_PRIO_PREEMPT(2)`)
    - Cada tarefa deve ser periódica (uso de `k_msleep()` ou `k_timer`).
    - Implementar e alternar entre os modos:
        - **Prioridade fixa preemptiva** (modo padrão).
        - **Round Robin** (entre tarefas de mesma prioridade).
        - **Simulação cooperativa** (tarefas se auto-suspendem com `k_yield()` ou `k_sleep()`).

4. **Instrumentação e Testes**
    - Adicionar logs via `printk()` com timestamp para rastrear a execução das tarefas.
    - Observar o comportamento em execução contínua e sob cargas artificiais.
    - Opcional: uso de ferramentas como SystemView, Tracealyzer (se disponível).
5. **Análise e Discussão**
    - Avaliar a latência, preempção, e regularidade de execução.
    - Comparar os modos de escalonamento e seus efeitos sobre tarefas com diferentes períodos.
    - Discutir implicações para aplicações reais.
6. **Relatório Técnico**
    - Apresentar a arquitetura do sistema.
    - Explicar a configuração do kernel (por exemplo, ativação do *time slicing*).
    - Incluir trechos de logs e análise dos dados coletados.
    - Relacionar os resultados com os conceitos estudados.

[DESCRIÇÃO – 11]
Desenvolvimento de Driver Android para Ativação da Lanterna com Temporizador

Desenvolver e testar um driver em ambiente Android que ative a lanterna do smartphone automaticamente após 30 segundos da inicialização do sistema, sem o uso de um aplicativo (APK), utilizando um simulador Android (como o Android Emulator do Android Studio ou AVD).

Normalmente, o controle da lanterna é realizado via aplicativos que utilizam a API de câmera. Neste projeto, o objetivo é criar uma solução a nível de sistema, implementando um driver ou módulo nativo para ativar a lanterna diretamente, integrando-se ao ciclo de inicialização do Android.

Ferramentas e Tecnologias

- **Android Emulator** ou **AVD Manager** (Android Studio);
- Código-fonte do Android (AOSP);
- Android NDK;
- Ferramentas ADB e Fastboot;
- Linguagens: C/C++, Shell Script e possivelmente Java;
- (Opcional) QEMU com Android embarcado.

Escopo do Projeto

1. **Análise do subsistema de câmera do Android**:
    - Estudo do controle da lanterna (flash LED) no HAL (Hardware Abstraction Layer);
    - Identificação dos pontos de entrada para ativar o hardware sem usar um app.
2. **Criação de um módulo de driver/modificação do HAL**:
    - Modificar ou estender o módulo da câmera para ativar o LED após 30s;
    - Utilizar `usleep()` ou `alarm()` para o temporizador.
3. **Integração com o processo de boot**:
    - Acionar o módulo com base em `init.rc` ou outro serviço via `init` do Android;
    - Iniciar o driver via script de inicialização ou serviço nativo.
4. **Emulação e Testes**:
    - Utilizar o Android Emulator com suporte a simulação de flash (configurações avançadas do AVD);
    - Monitorar logs com `logcat` para depuração;
    - Validar o funcionamento sem interação do usuário ou interface gráfica.

Etapas de Desenvolvimento e Testes

Etapa 1 – Configuração do Ambiente

- Instalar Android Studio com NDK e SDK.
- Baixar imagem Android compatível com suporte a câmera/flash no emulador.
- Clonar e compilar AOSP (se necessário).

Etapa 2 – Implementação do Driver

- Criar ou modificar o módulo responsável pelo controle do flash LED.
- Inserir lógica de temporizador (`sleep(30)` ou uso de `alarm/handler`).

- Garantir que a ativação da lanterna ocorra via chamada nativa (C/C++).

Etapa 3 – Integração com Sistema

- Configurar `init.rc` ou criar um serviço personalizado (`.rc` file);
- Testar boot do emulador e execução automática do driver.

Etapa 4 – Testes

- **Teste 1 – Ativação após Boot**: Verificar se a lanterna é ativada após 30s do boot.
- **Teste 2 – Logs de Debug**: Usar `logcat` para verificar mensagens do driver.
- **Teste 3 – Persistência**: Testar comportamento após reinício.
- (Opcional) **Teste em Dispositivo Real**: Caso possível, testar com dispositivo físico via ADB.

[DESCRIÇÃO – 12]
Construção de um Navegador Web Minimalista com Foco em Conceitos de Sistemas Operacionais

Este projeto tem como objetivo aplicar conceitos fundamentais de sistemas operacionais através da construção de um navegador web funcional e minimalista, utilizando softwares e bibliotecas já existentes. Os alunos devem compreender a interação entre o navegador, o sistema de arquivos, processos, gerenciamento de memória, rede e segurança, além de aspectos de usabilidade e testes.

Descrição:

Os alunos irão projetar e desenvolver um navegador web leve, baseado em bibliotecas de renderização e rede já consolidadas, como:

- **GTK WebKit** (para interface gráfica e motor de renderização),
- **Electron (Chromium + Node.js)** (caso optem por um navegador com mais funcionalidades e integração com o sistema),
- ou **Servo** (navegador experimental da Mozilla, escrito em Rust, para alunos mais avançados).

O foco não está na criação de um motor de renderização do zero, mas sim na **integração dos componentes**, na **personalização de funcionalidades**, e na **compreensão do papel do sistema operacional** em toda a arquitetura.

Etapas do Projeto:

1. **Pesquisa e Escolha da Stack Tecnológica:**
    - Avaliação de motores de renderização existentes (WebKit, Servo, Chromium).
    - Escolha da linguagem (C/C++, Rust, Python, JavaScript via Electron).

```
sudo apt update
sudo apt install build-essential git cmake libgtk-3-dev
libwebkit2gtk-4.0-dev
```

2. **Planejamento da Arquitetura do Navegador:**
    - Módulos: interface, renderização, rede, cache, histórico, abas.
    - Definição de como os processos/threads serão utilizados.
3. **Desenvolvimento do Protótipo Inicial:**
    - Criação da janela principal e carregamento básico de páginas. Exemplo:

```
#include <gtk/gtk.h>
#include <webkit2/webkit2.h>

int main(int argc, char *argv[]) {
    gtk_init(&argc, &argv);

    GtkWidget *window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    GtkWidget *webview = webkit_web_view_new();

    gtk_window_set_default_size(GTK_WINDOW(window), 800, 600);
    gtk_container_add(GTK_CONTAINER(window), webview);
    webkit_web_view_load_uri(WEBKIT_WEB_VIEW(webview), "https://example.com");

    g_signal_connect(window, "destroy", G_CALLBACK(gtk_main_quit), NULL);
    gtk_widget_show_all(window);

    gtk_main();
    return 0;
}
```

    - Integração com o sistema de arquivos para salvar histórico e favoritos. Exemplo:

```
#include <sqlite3.h>

sqlite3 *db;
sqlite3_open("historico.db", &db);
sqlite3_exec(db, "CREATE TABLE IF NOT EXISTS historico (url TEXT);", 0, 0, 0);
sqlite3_exec(db, "INSERT INTO historico (url) VALUES
('https://example.com');", 0, 0, 0);
sqlite3_close(db);
```

4. **Gerenciamento de Recursos e Processos:**
    - Criação de processos isolados para cada aba (modelo multiprocessado).
    - Monitoramento e controle de uso de CPU e memória.
5. **Recursos de Navegação:**
    - Implementação de botões de navegação, barra de endereços, bookmarks, histórico.
    - Suporte a múltiplas abas.
6. **Segurança e Sandboxing:**
    - Uso de técnicas de isolamento de processos e controle de permissões.
7. **Testes e Validação:**
    - Testes de desempenho (tempo de carregamento, consumo de CPU/RAM).

- Testes de estabilidade (abertura de múltiplas abas, fechamento forçado de processos).
- Testes de rede (respostas a requisições HTTP/HTTPS, manipulação de erros).
- Testes de compatibilidade com páginas populares.
- Testes de segurança (acesso não autorizado ao sistema de arquivos, isolamento entre abas).

8. **Documentação e Apresentação:**
    - Manual de uso do navegador.
    - Relatório técnico detalhando decisões de projeto, desafios e como o sistema operacional é utilizado.

[DESCRIÇÃO – 13]
Dashboard de Monitoramento de Servidores

Desenvolver um sistema web simplificado, inspirado em ferramentas como Grafana e Prometheus, que seja capaz de monitorar e exibir em tempo real métricas de desempenho de um servidor (Linux ou Windows). O foco é aplicar conceitos de sistemas operacionais como chamadas de sistema, gerenciamento de recursos e monitoração de processos.

Contexto Real

Empresas precisam garantir que seus servidores estejam funcionando corretamente e de forma eficiente. Para isso, é essencial monitorar continuamente o uso de CPU, memória RAM, disco e rede. Ferramentas como **Grafana**, **Prometheus** e **New Relic** são amplamente utilizadas no mercado.

Funcionalidades obrigatórias:

- Coletar métricas do sistema operacional local (uso de CPU, RAM, disco e rede).
- Armazenar os dados em um banco de dados leve (SQLite ou Redis).
- Visualizar métricas em tempo real via interface web com gráficos dinâmicos.
- Definir limites de alerta (ex: CPU > 80%) e exibir alertas visuais no painel.

Tecnologias Sugeridas

Backend (Escolher uma das opções):

- **Python + Flask + psutil**
    - `psutil` para coletar métricas do sistema.
    - `Flask` para fornecer uma API RESTful.
- **Node.js + os-utils** (ou módulo `os`, `systeminformation`)
    - API REST que envia as métricas coletadas para o frontend.

Banco de dados (um deles):

- **SQLite**: simples, baseado em arquivo, ideal para protótipos.
- **Redis**: rápido e eficiente, ideal para dados temporais e alertas.

Frontend:

- **HTML + JavaScript + Chart.js ou D3.js**
  - Painel de gráficos em tempo real.
  - Destaque visual quando limites forem ultrapassados.

Exemplo de Fluxo

1. Backend executa a cada N segundos coleta de métricas via chamadas de sistema.
2. Dados são armazenados no banco.
3. Frontend consome API REST para montar os gráficos.
4. Quando limites são ultrapassados, frontend exibe alerta (ex: barra vermelha ou popup).

Etapas de Desenvolvimento e Testes

Etapa	Descrição	Ferramenta Recomendada
1	Coleta de métricas locais	`psutil` (Python) ou `systeminformation` (Node.js)
2	Criação de API RESTful	Flask ou Express.js
3	Armazenamento de dados	SQLite ou Redis
4	Interface com gráficos em tempo real	Chart.js ou D3.js
5	Configuração de alertas	JS no frontend com condições visuais
6	Testes de uso intensivo (CPU, RAM)	Scripts de stress (`stress`, `stress-ng`)
7	Validação em diferentes SOs (Linux/Windows)	VirtualBox, WSL, etc.

1. Testes de Coleta de Métricas (Backend)

Objetivo: Validar se o sistema consegue obter corretamente as métricas do SO.

Procedimentos:

- Execute manualmente scripts que usam `psutil` (Python) ou `systeminformation` (Node.js).
- Verifique se os valores de CPU, RAM, disco e rede batem com os valores do `htop`, `top`, `taskmgr` (Windows) ou `free -m`.

Exemplo (Python):

```
python3 collect_metrics.py
```

Validação:

- Compare os valores de `psutil.cpu_percent()` com o `top` ou `htop`.
- Confirme que o sistema não retorna valores nulos ou negativos.

2. Testes de API REST (Integração Backend ↔ Banco de Dados)

Objetivo: Garantir que a API receba, processe e armazene corretamente os dados.

Procedimentos:

- Use o `Postman` ou `curl` para testar rotas da API, como `/metrics`.
- Envie dados simulados para a API e verifique se são persistidos corretamente no banco (SQLite ou Redis).

Exemplo (com `curl`):

```
curl http://localhost:5000/metrics
```

Validação:

- O JSON retornado deve conter as métricas corretas.
- O histórico deve crescer ao longo do tempo no banco.

3. Testes da Interface Web (Frontend)

Objetivo: Verificar se os gráficos estão funcionando e atualizando em tempo real.

Procedimentos:

- Acesse o painel web.
- Acompanhe se os gráficos são atualizados conforme as métricas mudam.
- Alterne o navegador (Chrome, Firefox) para verificar compatibilidade.

Validação:

- Gráficos com os dados corretos.
- Nenhum erro no console do navegador.
- Interface permanece responsiva e fluida.

4. Testes de Alerta de Limite

Objetivo: Confirmar que o sistema identifica e alerta o usuário quando os limites são ultrapassados.

Procedimentos:

- Use ferramentas como `stress` ou `stress-ng` para sobrecarregar o sistema.
- Configure limite de alerta, por exemplo, CPU > 80%.
- Observe se alerta visual (popup, cor, som) aparece corretamente.

Comando para gerar carga:

```
stress --cpu 2 --timeout 20
```

Validação:

- Alerta deve aparecer em até 2 segundos após ultrapassar o limite.
- Alerta deve desaparecer quando o valor retornar ao normal.

5. Testes de Persistência e Histórico

Objetivo: Garantir que os dados estejam sendo salvos e possam ser acessados posteriormente.

Procedimentos:

- Deixe o sistema rodando por alguns minutos.
- Verifique o banco de dados (SQLite: `DB Browser`, Redis: `redis-cli`) e analise se as métricas foram armazenadas com timestamp.

Validação:

- Cada registro deve conter valores e hora de coleta.
- O frontend deve conseguir exibir gráficos históricos.

6. Testes de Carga (Stress Test no Sistema Web)

Objetivo: Avaliar se o sistema suporta múltiplos acessos simultâneos.

Procedimentos:

- Use ferramentas como `Apache Benchmark` ou `locust.io`.
- Simule 10, 50 e 100 acessos simultâneos à API ou ao frontend.

Exemplo com Apache Benchmark:

```
ab -n 100 -c 10 http://localhost:5000/metrics
```

Validação:

- API deve responder sem queda de performance significativa.
- Nenhum erro 500 ou travamento da aplicação.

[DESCRIÇÃO – 14]
Simulador de Ataque DDoS com Mecanismos de Mitigação

Ataques de negação de serviço distribuída (DDoS) são uma ameaça constante à infraestrutura de servidores. Eles sobrecarregam sistemas com um alto volume de requisições falsas, tornando os serviços legítimos inacessíveis. Empresas utilizam soluções como **Cloudflare**, que atuam como **firewalls e balanceadores de carga**, para mitigar esses ataques.

Este projeto visa proporcionar uma experiência prática com os **conceitos fundamentais de sistemas operacionais**, como gerenciamento de redes, alocação de recursos e controle de processos, a partir da construção de um ambiente de simulação de ataques DDoS e implementação de defesas. Neste sentido, deve-se desenvolver um **laboratório web interativo** que:

1. Simule um ataque DDoS com múltiplos clientes gerando requisições em alta frequência (via `Socket.io`).

2. Visualize em tempo real a **carga do servidor** (RAM, CPU e conexões ativas).
3. Implemente mecanismos de mitigação como:
    - **Limite de requisições por IP**.
    - **Fila de requisições** com prioridade.
    - **Bloqueio temporário de IPs maliciosos** (banimento).
4. Analise o **impacto no desempenho do sistema operacional** em diferentes cenários de ataque.

Tecnologias Recomendadas:

Camada	Tecnologias
Backend	Node.js, Socket.io, Express.js
Frontend	HTML5, Chart.js, Bootstrap
Monitoramento	`os` module (Node.js), PM2, Netstat
Testes	Apache Benchmark (`ab`), Wrk

Funcionalidades Esperadas

- Painel Web com gráficos dinâmicos:
    - Quantidade de conexões simultâneas.
    - Consumo de CPU e RAM do processo Node.js.
    - IPs conectados em tempo real.
- Modo *Ataque*:
    - Clientes simulados enviam milhares de requisições por segundo.
    - Animação no frontend indicando sobrecarga.
- Modo *Defesa*:
    - Limitação de requisições por IP (ex: 20 reqs/s).
    - Visualização de IPs bloqueados.
    - Tempo de resposta médio antes/depois da mitigação.

Etapas de Desenvolvimento

1. **Planejamento e Configuração**
    - Escolha da arquitetura cliente-servidor.
    - Criação de ambiente de desenvolvimento com Node.js.
2. **Módulo de Ataque**
    - Simulação de múltiplos clientes via `Socket.io`.
    - Criação de scripts que geram tráfego automatizado (em loop ou threads).

3. **Módulo de Monitoramento**
    - Exposição de métricas via API REST (`/stats`).
    - Frontend com gráficos (Chart.js) atualizados por polling.
4. **Módulo de Mitigação**
    - Implementação de middleware no backend que:
        - Identifica IPs maliciosos.
        - Limita, adia ou bloqueia requisições.
    - Uso de cache simples com `Map()` para contagem de IPs.
5. **Testes e Avaliação**
    - Execução de ataques controlados com ferramentas (`ab`, `wrk`).
    - Coleta de métricas de estresse do sistema.
    - Comparação de desempenho com e sem mitigação.

Como o Projeto Deve Ser Testado

- **Cenário 1**: Servidor sem defesa.
    - Enviar 10.000 requisições simultâneas.
    - Medir tempo médio de resposta e consumo de recursos.
- **Cenário 2**: Servidor com limite de requisições.
    - Verificar queda de uso de CPU.
    - Validar IPs bloqueados e tempo de desbloqueio.
- **Cenário 3**: Ataque misto (clientes legítimos e maliciosos).
    - Medir eficiência do balanceamento de carga e da fila de requisições.

Utilizar ferramentas como:

- **`htop`** ou **`top`** para visualizar consumo de CPU/RAM.
- **`netstat`** para conexões abertas.
- **`ab -n 10000 -c 500 http://localhost:3000`** para simular ataques.

[DESCRIÇÃO – 15]
Simulação da Propagação de Vírus em Diferentes Sistemas de Arquivos

Investigar, por meio de simulações controladas, como vírus de computador podem se propagar em diferentes tipos de sistemas de arquivos (FAT32, NTFS, EXT4), explorando vulnerabilidades, mecanismos de persistência e execução automática.

Vírus e malwares utilizam características específicas dos sistemas de arquivos para se esconder, se replicar ou se autoexecutar. Por exemplo, em sistemas FAT32, é comum o uso de arquivos ocultos com execução automática (autorun.inf), enquanto no EXT4 ou NTFS, malwares podem explorar

permissões, links simbólicos, e atributos especiais.

Softwares e Ferramentas Utilizadas

- **QEMU** ou **VirtualBox**: para criar ambientes isolados com diferentes sistemas operacionais e sistemas de arquivos.
- **mkfs**: para formatar partições em FAT32, EXT4, etc.
- **TestDisk** ou **GParted**: para manipulação de partições de forma visual.
- **Python** ou **Bash scripts**: para simular "vírus" de teste (ex: cópia automática, criação de arquivos ocultos, simulação de infecção).
- **ClamAV**: antivírus open-source para testes de detecção.
- **Samba/USB Image**: para simular compartilhamento de arquivos infectados ou dispositivos removíveis.

Etapas do Projeto

1. Introdução e Estudo

- Estudo das principais características de sistemas de arquivos: FAT32, NTFS e EXT4.
- Estudo básico sobre malwares: tipos, vetores de propagação, técnicas de persistência.

2. Montagem dos Ambientes de Teste

- Criação de 3 máquinas virtuais com os seguintes sistemas de arquivos:
  - VM1: Linux com EXT4
  - VM2: Windows com NTFS
  - VM3: Linux ou Windows com FAT32 em dispositivo removível (pendrive virtual)
- Instalação de ferramentas auxiliares (ClamAV, editores hexadecimais, etc.)

3. Desenvolvimento dos Simuladores de Vírus

- Scripts que executam ações típicas de malware:
  - Copiam-se automaticamente para novas mídias.
  - Criam arquivos ocultos.
  - Alteram permissões ou atributos especiais.
  - Simulam execução automática (ex: autorun.inf em FAT32).

Exemplo:

```
# Simula a infecção
def infect_drive(drive_path):
    try:
        shutil.copy("virus_simulado.bat", os.path.join(drive_path, "virus_simulado.bat"))
        shutil.copy("autorun.inf", os.path.join(drive_path, "autorun.inf"))
        print(f"[+] Dispositivo infectado: {drive_path}")
    except Exception as e:
```

```
print(f"[-] Erro ao infectar {drive_path}: {e}")
```

4. Execução dos Casos de Teste

- Inserção do “vírus” no ambiente e monitoramento:
  - Qual sistema de arquivos permite execução automática?
  - Onde o “vírus” permanece invisível ou persistente?
  - Como a detecção varia entre os sistemas?

5. Análise de Resultados

- Comparação entre os sistemas de arquivos:
  - Facilidade de infecção.
  - Facilidade de detecção.
  - Técnicas eficazes de mitigação.

🔬 Testes e Validação

Cada cenário de simulação deve ser testado com:

- Execução do script "vírus" em diferentes sistemas de arquivos.
- Monitoramento com antivírus (ClamAV) e logs do sistema.
- Comparação de resultados quanto à eficácia da propagação.
- Restauração do sistema a partir de snapshot para recomeçar o teste.

[DESCRIÇÃO – 15] Modelagem de Deadlock em Sistema de Impressão Corporativo com Redes de Petri

Desenvolver e analisar a modelagem de um sistema de impressão em uma rede corporativa onde múltiplos processos competem por recursos limitados (impressoras e memória). O projeto visa demonstrar, por meio de Redes de Petri, como deadlocks podem ocorrer e como evitá-los.

Em ambientes corporativos, múltiplos usuários enviam documentos para impressoras compartilhadas. Cada processo de impressão pode exigir uma impressora *e* uma quantidade de memória para armazenar o spool do documento. Se os recursos forem alocados em ordem incorreta, pode ocorrer **deadlock**.

Exemplo Clássico de Deadlock:

- Processo A aloca uma **impressora** e espera por **memória**.
- Processo B aloca **memória** e espera por uma **impressora**.
- Ambos ficam bloqueados indefinidamente.

Etapas do Projeto:

1. **Levantamento de Requisitos:**

- Definir o número de processos concorrentes.
- Definir o número de impressoras e blocos de memória disponíveis.

2. **Modelagem com Rede de Petri:**
   - Lugares: `Processos`, `Impressoras livres`, `Memória livre`.
   - Transições: `Alocar Impressora`, `Alocar Memória`, `Liberar Recursos`.
   - Estados iniciais: Todos os recursos livres, processos prontos.
3. **Simulação Visual:**
   - Usar **CPN Tools, PIPE, SNOOPY** para modelar e simular:
     - Cenários sem deadlock.
     - Cenários com deadlock (hold-and-wait).
   - Gerar árvore de marcações para identificar ciclos de espera.
4. **Simulação via Código:**
   - Reproduzir o modelo em **Python** com `snakes` ou `PyPN`.
   - Automatizar testes de múltiplos cenários de alocação.
5. **Verificação Formal:**
   - Calcular **invariantes de lugar** para verificar conservação de recursos.
   - Demonstrar por análise estrutural ou árvore de alcance a ocorrência de deadlock.
6. **Proposição de Soluções:**
   - Reordenar alocação de recursos (ex: alocar sempre impressora *depois* da memória).
   - Implementar **algoritmos de prevenção** (ex: Banker's Algorithm simplificado).
   - Propor **políticas de timeout ou rollback**.

[DESCRIÇÃO – 16]
Verificação de Data Races em Programas Concorrentes via SAT Solver

Investigar se o uso de mecanismos de sincronização (locks, mutexes, semáforos) é suficiente para evitar *data races* em programas concorrentes, por meio de verificação formal utilizando lógica proposicional e resolução com SAT Solvers.

Em ambientes com múltiplas *threads*, o uso incorreto de locks pode levar a *data races*, onde duas ou mais threads acessam uma mesma variável simultaneamente, e pelo menos uma dessas acessa para escrita. Este projeto propõe o uso de técnicas formais, modelagem de estados e resolução lógica para detectar automaticamente possíveis *data races* em programas concorrentes.

Ferramentas Recomendadas

- **Linguagem:** C/C++ (com pthreads) ou Rust (com `std::sync`)
- **SAT Solver:** MiniSAT ou Z3 SMT Solver

- **Analisador de Código:** Clang (para análise de AST), CIL, Frama-C ou ferramenta de parsing leve como `pycparser`
- **Ambiente de Testes:** Linux com suporte a *threading*

Etapas do Projeto

1. Análise Estática de Locks

- Analisar o código fonte e identificar onde locks são adquiridos e liberados.
- Extrair por função e thread:
  - Variáveis globais acessadas.
  - Locks/mutexes associados a esses acessos.

2. Modelagem do Programa como Sistema de Transições

- Representar o comportamento das threads como uma sequência de ações:
  - `lock(m)`, `unlock(m)`, `read(x)`, `write(x)`
- Modelar os estados globais e transições permitidas entre eles.
- Criar uma árvore de possíveis interleavings com base em essas transições.

3. Codificação de Regras de Data Race

- Transformar as regras de acesso concorrente sem proteção em fórmulas booleanas.
- Exemplo:

```
write(x)_T1 ∧ read(x)_T2 ∧ ¬(locked_by_same_mutex(x)_T1 ∧ locked_by_same_mutex(x)_T2)
```

- Gerar proposições para cada acesso e sua relação com os locks.

4. Resolução com SAT Solver

- Codificar as proposições no formato CNF (Conjuntiva Normal Form).
- Alimentar o SAT Solver (ex: MiniSAT) com a representação.
- Verificar satisfatibilidade:
  - **SAT** → Há uma execução onde o data race ocorre.
  - **UNSAT** → O uso de locks é suficiente para evitar o data race.

5. Extração de Contraexemplos

- Se a instância for satisfatível, extrair a sequência de ações que leva ao data race.
- Representar graficamente ou em texto o caminho de execução onde a falha ocorre.

Sugestão de Testes

- Códigos com falhas conhecidas (e.g., sem lock em uma thread).
- Códigos com sincronização correta usando `pthread_mutex`.
- Códigos com *locks* aninhados ou múltiplos recursos.

Exemplo das etapas:

Código em C (com `pthread`)

```
#include <pthread.h>
#include <stdio.h>

int x = 0;

void* thread1(void* arg) {
    x = x + 1;  // Escrita sem proteção
    return NULL;
}

void* thread2(void* arg) {
    int y = x;  // Leitura sem proteção
    return NULL;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return 0;
}
```

Transições Extraídas

Thread	Ação	Objeto	Tipo
T1	write(x)	x	write
T2	read(x)	x	read

Sem uso de `pthread_mutex_lock/unlock`, portanto não há proteção explícita.

Codificação SAT (simplificada)

Vamos representar:

- `W1`: `write(x)` na T1
- `R2`: `read(x)` na T2
- `P(W1, R2)`: acesso protegido por lock comum → **false**
- `Race(W1, R2)`: data race entre W1 e R2

Fórmula para detectar race:

```
Race(W1, R2) ⇔ W1 ∧ R2 ∧ ¬P(W1, R2)
```

Substituindo:

```
Race(W1, R2) ⇔ true ∧ true ∧ ¬false = true
```

Resultado: SAT → Existe *data race*.

Corrigindo o Código

```
#include <pthread.h>
#include <stdio.h>

int x = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* thread1(void* arg) {
    pthread_mutex_lock(&lock);
    x = x + 1;
    pthread_mutex_unlock(&lock);
    return NULL;
}

void* thread2(void* arg) {
    pthread_mutex_lock(&lock);
    int y = x;
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

Agora, temos:

- `P(W1, R2) = true` (ambos protegidos pelo mesmo lock)

```
Race(W1, R2) ⇔ true ∧ true ∧ ¬true = false
```

Resultado: UNSAT → Sem data race.

Como Rodar com MiniSAT

1. Gerar fórmula CNF do problema:

```
p cnf 3 2
1 2 0          ; W1 ∧ R2
-3 0           ; ¬P(W1, R2)
```

(onde var 1 = W1, 2 = R2, 3 = P(W1, R2))

2. Rodar o solver: `minisat race.cnf output.txt`
3. Interpretar a saída:

- SAT: existe *data race*
- UNSAT: não existe